University of Edinburgh, School of Informatics

Secure Programming Coursework: Part 2

3. Secure Server Management

3.1 OpenSSH Configuration

a. To allow only `user` login to the server via ssh I added the following configuration in the `/etc/ssh/sshd_config` file.
Configuration: `AllowUsers user`.

b. To allow only `user` login to the server using only ssh private key I did the following:

1. Generated an RSA public-private key pair using this command:
   `ssh-keygen -t rsa`
2. Copied my public key to the server using this command:
   `ssh-copy-id -p2222 user@localhost`. The file `authorized_keys` will be updated accordingly.
3. Added the following configurations to the `/etc/ssh/sshd_config` file.
   Configuration: `AuthenticationMethods publickey`.
   Configuration: `PasswordAuthentication no`.

3.2 Fun with Heartbleed

The Heartbleed is a vulnerability in the OpenSSL library used on a significant number of servers. Essentially this vulnerability allows attackers to steal information intended to be protected by SSL/TLS encryption. Notably, the bug regards the heartbeat request between a client and a host. The OpenSSL heartbeat extension allows either end-point of a TLS connection to detect whether its peer is still present. A HeartbeatRequests message consists of a one-byte type field, a two-byte payload length field, a payload and at least 16 bytes of padding. An adversary can exploit this by sending a message with a larger payload length than the HearbeatRequest message. Then the server responds with up to 2^16 bytes of memory leaking information. The relevant attack on Hearthbleed is CVE-2014-0160, and the possible attack consequences include the leak of server private keys, server's users' data etc. It is important to mention that patching the vulnerability on a host is not enough since its certificate could be compromised. Thus it is a recommended to revoke and replace the host's certificate using a new one generated from a different private key.

To patch the Heartbleed vulnerability we did the following:

1. Run the OpenSSL service.
2. Use `nmap` to scan for the vulnerability.
   `nmap -p 54321 --script ssl-heartbleed localhost`
   ```
   Nmap scan report for localhost (127.0.0.1)
   Host is up (0.00014s latency).
   Other addresses for localhost (not scanned): ::1
   PORT      STATE SERVICE
   54321/tcp open  unknown
   | ssl-heartbleed:
   |   VULNERABLE:
   |   The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic softwa
   |     State: VULNERABLE
   |     Risk factor: High
   |       OpenSSL versions 1.0.1 and 1.0.2-beta releases (including 1.0.1f and 1.0.2-beta1) of
   |
   |     References:
   |       https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
   |       http://www.openssl.org/news/secadv_20140407.txt
   |_      http://cvedetails.com/cve/2014-0160/
   ```
3. Located heartbeat related files in the library using the command:
   `find . -name '*.c' | xargs grep heartbeat`.
4. Isolated the two related files which are `t1_lib.c` and `d1_both.c`.
5. The applied fix was to check if the heartbeat type, length, payload and padding was in certain boundaries. If the sum of it was greater than the record length we discard the response.
   ```
   if (1 + 2 + 16 > s->s3->rrec.length)
          return 0;
   ```
6. We cross checked the patch referencing to the original fix on Github
7. Recompiled the library and scaned using `nmap`.
   `nmap -p 54321 --script ssl-heartbleed localhost`

```
Starting Nmap 7.92 ( https://nmap.org ) at 2022-03-13 20:19 GMT
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00020s latency).
Other addresses for localhost (not scanned): ::1

PORT      STATE SERVICE
54321/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 19.22 seconds
```

3.3 Digital Signature Service

Next we provide 16 security modifications and good programming practices we implmented in our web app. We also state possible vulnerabilities and how our approach mitigates them. When realizing the application we considered web app common flaws, including SQL-Injections, XSS, Insecure access to resources, Server missconfigurations, and sensitive data exposure. We do not include full code snippets in this report. The relevant code can be found in the submited code repository.

1. **Configurations**
   - **Operating System**
     - Change the permissions of db/ds_service.db so only the user `owner` of the file has full access on it, while `group` and `others` can only read the file. We make this modification using `sudo chmod 644 db/ds_service.db`.
     - Created a new functions file in the `/include/my_functions` directory. The new file is not under `root` user.
   - **Database configurations**
     - We created a new table with a slightly different schema to accomodate two more columns where we save the RSA public and private keys. We use the following command to create a new table named `users_new`: `CREATE TABLE users_new (username TEXT PRIMARY KEY, password`
   - **Server configurations**
     - Any website visitor can browse the server's directories typing the correct url. We want to avoid that since it might leak information about the web app intrinsics. We

restrict access on URL accessed objects updating the `/etc/httpd/conf/httpd.conf` file. We added the following configurations so an attacker is not able to download the database file or access the `my_functions.php` file:

```
<Location "/include/my_functions.php">
    Order Allow,Deny
    Deny from all
</Location>

<Location "/db">
    Order Allow,Deny
    Deny from all
</Location>
```

    - We have noticed that the server uses a vulnerable version of the OpenSSL library `OpenSSL/1.0.1f`. However, after scanning the server using a tool described in the previous section `3.2 Fun with Heartbleed`, we found no issues. We don't know if the server is configured to use the fixed library.
- **App configurations**
    - We enabled openssl functionality on our app updating the `/etc/php/php.ini` file located. We uncommented line 888 (`extension=openssl.so`).
    - When inspecting the request response headers we can view the server and php versions. This can leak information that help an attacker prepare an exploit. We switched the `expose_php` option in the `/etc/php/php.ini` file to `Off`.
    - We deleted the `include/.admin.php` since it can leak information about the server version and php version. This was an infomation leakage vulnerability. Even though the access to this file is restricted by server level configurations, we assumed that it is a good practice to remove it.
    - We also reduced the `post_max_size` from `128MB` to

`12KB` (in `/etc/php/php.ini`). The message can be 10K characters while the public key and signature are significantly smaller. By limiting the post size, we protect the system from possible repeated requests that aim to waste resources.

2. **Web Application implementation details**
    - The provided code in `functions.php` is vulnerable to SQL-Injections since the variable binding is not done correctly. We provide a mitigation to this problem in `my_functions.php` where we use the `bindParam()` instuction to safely assign user input to our sql queries.
    - The cookie approach used in the provided code is not secure since cookies can be manipulated by the user. Further, the cookie value was generated hashing username using MD5. MD5 is a broken algorithm, thus not a secure sheme to use. We mitigate these issues by implementing a basic `CSRF` token mechanism, where each token stays alive for 3600 seconds. We implemented the functions `create_csrf_token()` to create a token when a user logs in and `validate_csrf_token()` to check if the token is valid when the user interacts with the website. When the user logs out we unset the csrf token and the user from the session.
    - The user might input invalid malicious input trying to perform an XSS attack. To avoid that, the input is validated. We implement a function `validate()` that uses the php provided instruction `htmlspecialchars()`. We also, `trim()` the input and apply `stripslashes()` to remove `\` that might trigger unexpected execution results.
    - Sometimes displaying errors on the user side is not a wise choice since this can leak information about the code intrinsics. Hence, we removed the `ini_set('display_errors', On);` from the given `index.php` file.
    - A less likely, yet possible scenario is a data breach and compromise of our database. Therefore, storing plain text passwords on it is not a good prictice. To avoid this, we apply hashing `password_hash($password, PASSWORD_DEFAULT);` to securely

represent passwords. When a user logs in we compare hashes and not the original passwords.

- As mentioned before plain text stores on the database is not a good prictice. Therefore, we store private keys encrypted. Each private key is encrypted using a different key, to enhance security. We though that a simple solution is to use the username of each account. A better solution that was not implemented, is to use a passphrase that only the user knows, to encrypt and decrypt the private key. The encryption algorithm used is `AES-256-CBC`.
- We check if the incoming request is initiated from a form submition within the interface and not using an external tool such as `curl`. This can slow down the attackers but is not an essential mitigation.